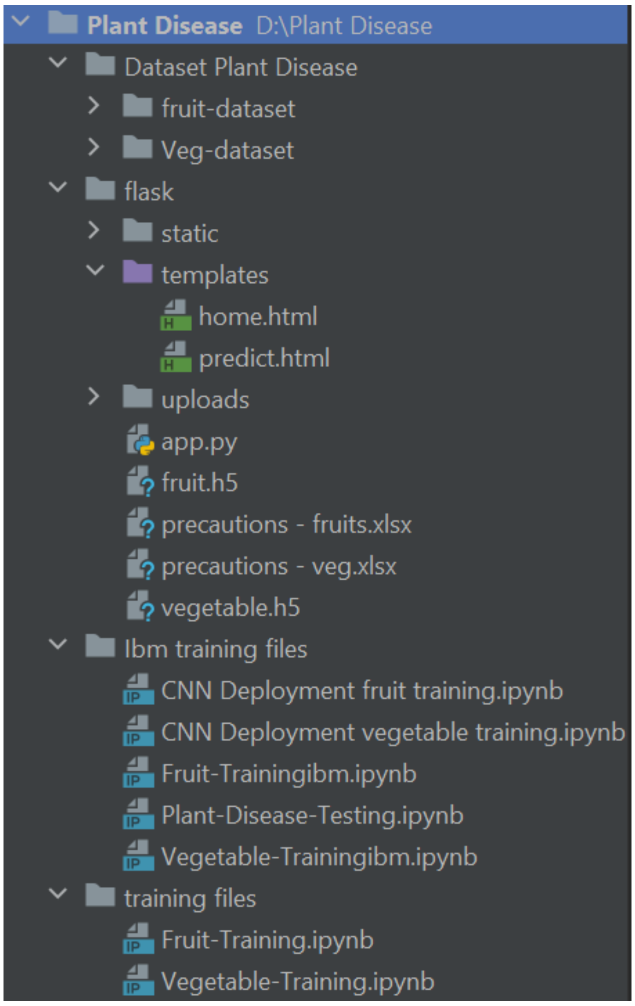
Project Structure



```
Plant Disease  D:\Plant Disease
    Dataset Plant Disease
        fruit-dataset
        Veg-dataset
    flask
        static
        templates
            home.html
            predict.html
        uploads
        app.py
        fruit.h5
        precautions - fruits.xlsx
        precautions - veg.xlsx
        vegetable.h5
    Ibm training files
        CNN Deployment fruit training.ipynb
        CNN Deployment vegetable training.ipynb
        Fruit-Trainingibm.ipynb
        Plant-Disease-Testing.ipynb
        Vegetable-Trainingibm.ipynb
    training files
        Fruit-Training.ipynb
        Vegetable-Training.ipynb
```

- The dataset folder contains two folders for the fruit and vegetable dataset which again contains a test and train folder, each of them have images of different diseases.
- The Flask folder has all the files necessary to build the flask application.
    - the static folder has the images, style sheets, and scripts that are needed in building the web page.
    - templates folder has the HTML pages.
    - uploads folder has the uploads made by the user.
    - app.py is the python script for server-side computing.
    - .h5 files are the model files that are to be saved after model building.
    - precautions excel files contain the precautions for all kinds of diseases.
- Fruit-Training.ipynb, Vegetable-Training, and Plant-Disease-Testing.ipynb are the training and testing notebooks.

- IBM folder contains IBM deployment files.

Data Collection

The first step is to download the dataset

Create Train and Test folders with each folder having subfolders with leaf images of different plant diseases. You can collect datasets from different open sources like kaggle.com, data.gov, UCI machine learning repository, etc. The folder contains the provided in the project structure section has the link from where you can download datasets that can be used for training. Two datasets will be used, we will be creating two models one to detect vegetable leaf diseases like tomato, potato, and pepper plants and the second model would be for fruits diseases like corn, peach, and apple.

Download Dataset

Download the Plant Disease dataset.
https://drive.google.com/file/d/1fxs7ptI6zh7NTbCOZARKZ7AmYKjnprrY/view?usp=sharing

Image Preprocessing

Now that we have all the data collected, let us use this data to train the model . before training the model you have to preprocess the images and then feed them on to the model for training. We make use of Keras ImageDataGenerator class for image preprocessing.
For more info about image preprocessing please click on the below link
data Augmentation

Image Pre-processing includes the following main tasks

- Import ImageDataGenerator Library.
- Configure ImageDataGenerator Class.
- Applying ImageDataGenerator functionality to the trainset and test set.

Note: The ImageDataGenerator accepts the original data, randomly transforms it, and returns only the new, transformed data.
To know more about the data generator class click on this **link**
Lets build model for fruit leaf disease detection

Open Jupyter notebook and create a new python file, name ii Fruit-Training.ipynb and save it in the project folder. To know more about the usage of the Jupyter notebook watch the video given in the pre-requisites section

Preprocess The Images

Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset.

The Keras deep learning neural network library provides the capability to fit models using image data augmentation via the ImageDataGenerator class.

The first step is usually importing the libraries that will be needed in the program.

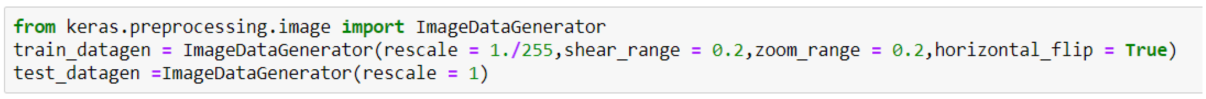
Import Keras library from that library import the ImageDataGenerator Library to your Python script:

After each code block in this tutorial, you should type ALT + ENTER or SHIFT+ENTER to run the code and move into a new code block within your notebook.

Let us import the ImageDataGenerator class from Keras

Import ImageDataGenerator Library and Configure it

ImageDataGenerator class is used to load the images with different modifications like considering the zoomed image, flipping the image and rescaling the images to range of 0 and 1.



```
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1./255,shear_range = 0.2,zoom_range = 0.2,horizontal_flip = True)
test_datagen =ImageDataGenerator(rescale = 1)
```

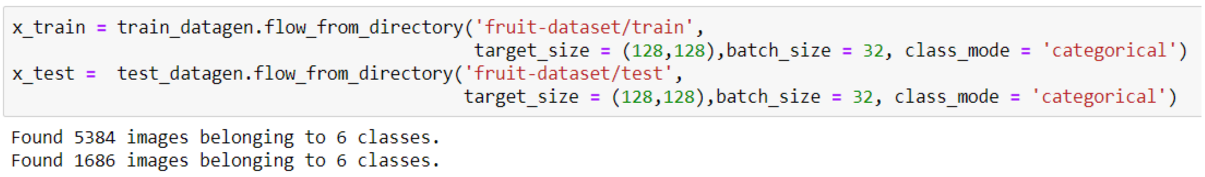
There are five main types of data augmentation techniques for image data; specifically:

- Image shifts via the width_shift_range and height_shift_range arguments.
- The image flips via the horizontal_flip and vertical_flip arguments.
- The image rotates via the rotation_range argument
- Image brightness via the brightness_range argument.
- The image zooms via the zoom_range argument.

An instance of the ImageDataGenerator class can be constructed for train and test.

Apply ImageDataGenerator functionality to Train and Test set

Specify the path of both the folders in the flow_from_directory method. We are importing the images in 128*128 pixels.



```
x_train = train_datagen.flow_from_directory('fruit-dataset/train',
                                            target_size = (128,128),batch_size = 32, class_mode = 'categorical')
x_test =  test_datagen.flow_from_directory('fruit-dataset/test',
                                           target_size = (128,128),batch_size = 32, class_mode = 'categorical')
```

```
Found 5384 images belonging to 6 classes.
Found 1686 images belonging to 6 classes.
```

To know more about this method *click here*

Model Building For Fruit Disease Prediction

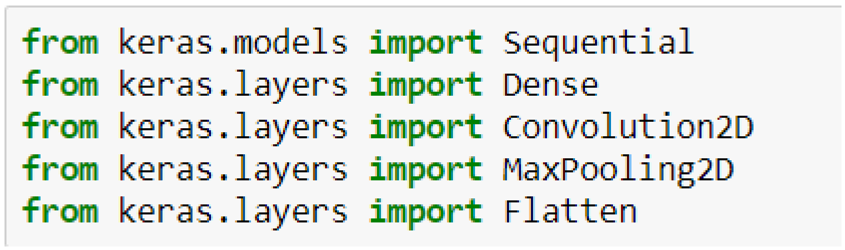
We are ready with the augmented and pre-processed image data, Lets begin our model building, this activity includes the following steps

- Import the model building Libraries
- Initializing the model
- Adding CNN Layers
- Adding Hidden Layer
- Adding Output Layer
- Configure the Learning Process
- Training and testing the model
- Saving the model

To know more about model building please **click here**

Import The Libraries

Import the libraries that are required to initialize the neural network layer, and create and add different layers to the neural network model.



```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Convolution2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
```

Initializing The Model

Keras has 2 ways to define a neural network:

- Sequential
- Function API

The Sequential class is used to define linear initializations of network layers which then, collectively, constitute a model. In our example below, we will use the Sequential constructor to create a model, which will then have layers added to it using the add () method.

Now, will initialize our model.

Initialize the neural network layer by creating a reference/object to the Sequential class.

```
model=Sequential()
```

ADD CNNLayers

We will be adding three layers for CNN

- Convolution layer
- Pooling layer
- Flattening layer

Add Convolution Layer

The first layer of the neural network model, the convolution layer will be added. To create a convolution layer, Convolution2D class is used. It takes a number of feature detectors, feature detector size, expected input shape of the image, and activation function as arguments. This layer applies feature detectors on the input image and returns a feature map (features from the image).

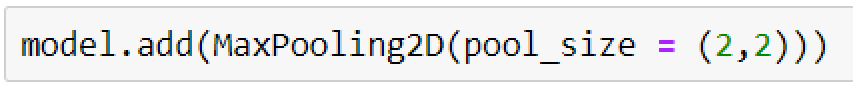
Activation Function: These are the functions that help us to decide if we need to activate the node or not. These functions introduce non-linearity in the networks.

```
model.add(Convolution2D(32,(3,3),input_shape = (128,128,3),activation = 'relu'))
```

Add the pooling layer

Max Pooling selects the maximum element from the region of the feature map covered by the filter. Thus, the output after the max-pooling layer would be a feature map containing the most prominent features of the previous feature map.

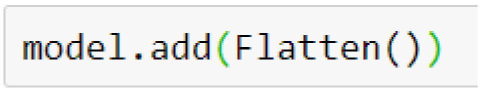
After the convolution layer, a pooling layer is added. Max pooling layer can be added using MaxPooling2D class. It takes the pool size as a parameter. Efficient size of the pooling matrix is (2,2). It returns the pooled feature maps. (Note: Any number of convolution layers, pooling and dropout layers can be added)



```
model.add(MaxPooling2D(pool_size = (2,2)))
```

Add the flatten layer

The flatten layer is used to convert n-dimensional arrays to 1-dimensional arrays. This 1D array will be given as input to ANN layers.



```
model.add(Flatten())
```

To know more about the convolutional layers, refer to this **link**

Add Dense Layers

Now, let's add Dense Layers to know more about dense layers click below

Dense layers

The name suggests that layers are fully connected (dense) by the neurons in a network layer. Each neuron in a layer receives input from all the neurons present in the previous layer. Dense is used to add the layers.

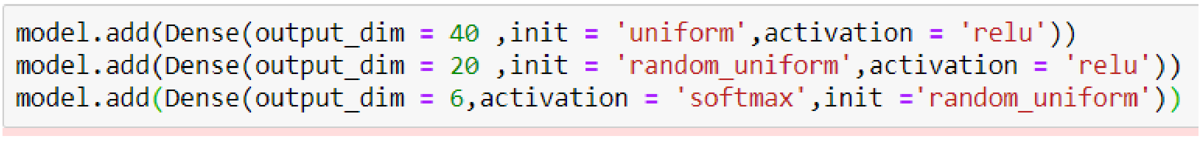
Adding Hidden layers

This step is to add a dense layer (hidden layer). We flatten the feature map and convert it into a vector or single dimensional array in the Flatten layer. This vector array is fed it as an input to the neural network and applies an activation function, such as sigmoid or other, and returns the output.

- init is the weight initialization; function which sets all the weights and biases of a network to values suitable as a starting point for training.
- units/ output_dim, which denote is the number of neurons in the hidden layer.
- The activation function basically decides to deactivate neurons or activate them to get the desired output. It also performs a nonlinear transformation on the input to get better results on a complex neural network.
- You can add many hidden layers, in our project we are added two hidden layers. The 1st hidden layer with 40 neurons and 2nd hidden layer with 20neurons.

Adding the output layer

This step is to add a dense layer (output layer) where you will be specifying the number of classes your dependent variable has, activation function, and weight initializer as the arguments. We use the add () method to add dense layers. the output dimensions here is 6



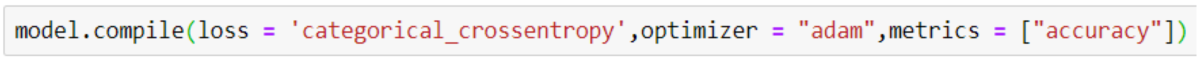
```
model.add(Dense(output_dim = 40 ,init = 'uniform',activation = 'relu'))
model.add(Dense(output_dim = 20 ,init = 'random_uniform',activation = 'relu'))
model.add(Dense(output_dim = 6,activation = 'softmax',init ='random_uniform'))
```

Note: if you have only one or two classes in the output layer, assign "units= 1" and "activation = sigmoid". If you have more than two classes (let's assume 3) then assign "units / output_dim = 3" and "activation = softmax".

Train And Save The Model

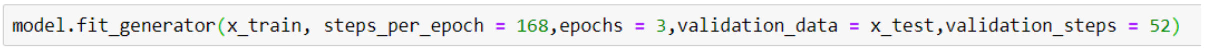
Compile the model

After adding all the required layers, the model is to be compiled. For this step, loss function, optimizer and metrics for evaluation can be passed as arguments.



```
model.compile(loss = 'categorical_crossentropy',optimizer = "adam",metrics = ["accuracy"])
```

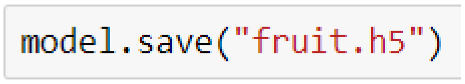
Fit and save the model
Fit the neural network model with the train and test set, number of epochs and validation steps. Steps per epoch is determined by number of training images//batch size, for validation steps number of validation images//batch size.



```
model.fit_generator(x_train, steps_per_epoch = 168,epochs = 3,validation_data = x_test,validation_steps = 52)
```

Accuracy, Loss: Loss value implies how poorly or well a model behaves after each iteration of optimization. An accuracy metric is used to measure the algorithm's performance in an interpretable way. The accuracy of a model is usually determined after the model parameters and is calculated in the form of a percentage.

The weights are to be saved for future use. The weights are saved in as .h5 file using save().



```
model.save("fruit.h5")
```

model.summary() can be used to see all parameters and shapes in each layer in our models.

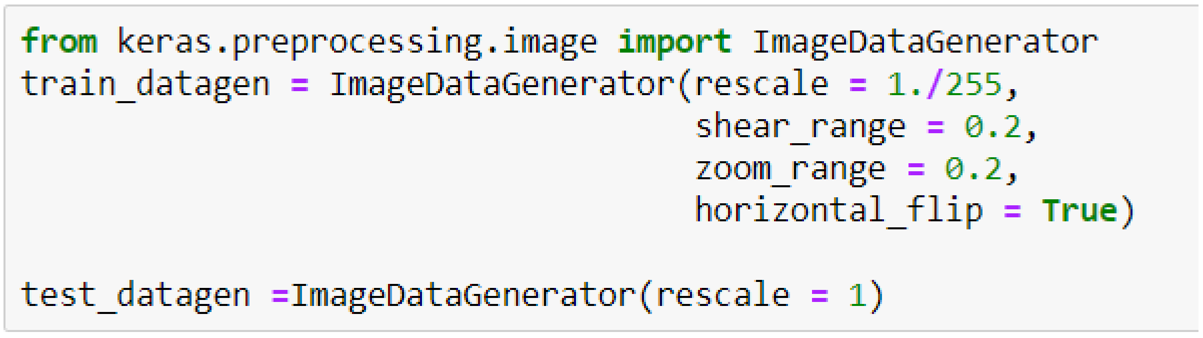
Model Building For Vegetable Disease Prediction

Create an other jupyter notebook file in the project folder and name it as vegitable_training. The same steps followed for the fruit disease prediction model are to be followed to train the tomato, potato, and pepper diseases.

Train And Save The Model

Import ImageDataGenerator Library and Configure it

ImageDataGenerator class is used to load the images with different modifications like considering the zoomed image, flipping the image and rescaling the images to range of 0 and 1.



```
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   shear_range = 0.2,
                                   zoom_range = 0.2,
                                   horizontal_flip = True)

test_datagen =ImageDataGenerator(rescale = 1)
```

Apply ImageDataGenerator functionality to Train and Test set

Specify the path of both the folders in the flow_from_directory method. We are importing the images in 128*128 pixels.

```
x_train = train_datagen.flow_from_directory('Veg-dataset/train_set',
                                            target_size = (128,128),
                                            batch_size = 16,
                                            class_mode = 'categorical')
```

```
Found 11386 images belonging to 9 classes.
```

```
x_test = test_datagen.flow_from_directory('Veg-dataset/test_set',
                                          target_size = (128,128),
                                          batch_size = 16,
                                          class_mode = 'categorical')
```

```
Found 3416 images belonging to 9 classes.
```

Import the required model building libraries

Import the libraries that are required to initialize the neural network layer, create and add different layers to the neural network model.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Convolution2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten
```

initialize the model

Initialize the neural network layer by creating a reference/object to the Sequential class.

```
model=Sequential()
```

Add the convolution layer

In the first layer of the neural network model, the convolution layer will be added. To create a convolution layer, the Convolution2D class is used. It takes a number of feature detectors, features detector size, expected input shape of the image, and activation function as arguments. This layer applies feature detectors on the input image and returns a feature map (features from the image).

Activation Function: These are the functions that help us to decide if we need to activate the node or not. These functions introduce non-linearity in the networks.

```
model.add(Convolution2D(32,(3,3),input_shape = (128,128,3),activation = 'relu'))
```

Add the pooling layer

Max Pooling selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.

After the convolution layer, a pooling layer is added. Max pooling layer can be added using MaxPooling2D class. It takes the pool size as a parameter. Efficient size of the pooling matrix is (2,2). It returns the pooled feature maps. (Note: Any number of convolution layers, pooling and dropout layers can be added)

```
model.add(MaxPooling2D(pool_size = (2,2)))
```

Add the flatten layer

The flatten layer is used to convert n-dimensional arrays to 1-dimensional arrays. This 1D array will be given as input to ANN layers.

```
model.add(Flatten())
```

Adding the dense layers

Three dense layers are added which usually take a number of units/neurons. Specifying the activation function, kind of weight initialization is optional.

```
model.add(Dense(units = 300, init ='uniform', activation ='relu'))
```

```
model.add(Dense(units = 150, init ='uniform', activation ='relu'))
```

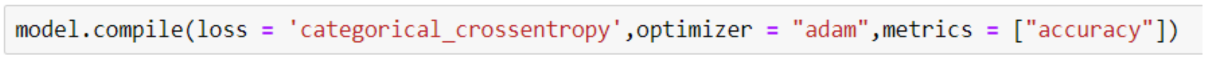
```
model.add(Dense(units = 75, init ='uniform', activation ='relu'))
```

```
model.add(Dense(output_dim = 9,activation = 'softmax',init ='uniform'))
```

Note: Any number of convolution, pooling, and dense layers can be added according to the data.

Compile the model

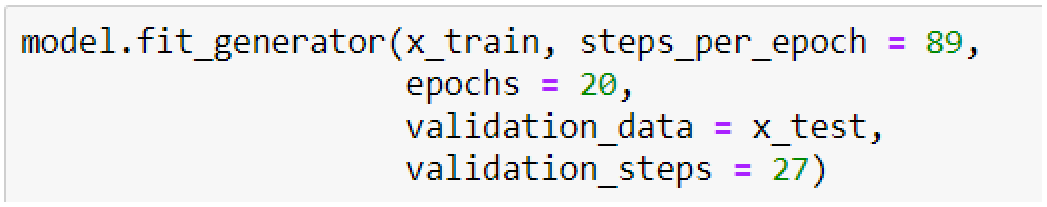
After adding all the required layers, the model is to be compiled. For this step, loss function, optimizer and metrics for evaluation can be passed as arguments.



```
model.compile(loss = 'categorical_crossentropy',optimizer = "adam",metrics = ["accuracy"])
```

Fit and save the model

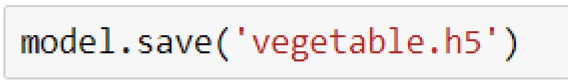
Fit the neural network model with the train and test set, number of epochs and validation steps. Steps per epoch is determined by number of training images//batch size, for validation steps number of validation images//batch size.



```
model.fit_generator(x_train, steps_per_epoch = 89,
                    epochs = 20,
                    validation_data = x_test,
                    validation_steps = 27)
```

Accuracy, Loss: Loss value implies how poorly or well a model behaves after each iteration of optimization. An accuracy metric is used to measure the algorithm's performance in an interpretable way. The accuracy of a model is usually determined after the model parameters and is calculated in the form of a percentage.

The weights are to be saved for future use. The weights are saved in as .h5 file using save().



```
model.save('vegetable.h5')
```

model.summary() can be used to see all parameters and shapes in each layer in our models.